Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems

Juhnyoung Lee and *Sang H. Son*

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

Studies in [7, 8, 9] concluded that for a variety of reasons, optimistic concurrency control appears well-suited to real-time database systems. Especially, they showed that in a real-time database system that discards tardy transactions, optimistic concurrency control outperforms locking. In this paper, we show that the optimistic algorithms used in those studies incur restarts unnecessary to ensure data consistency. We present a new optimistic concurrency control algorithm that can avoid such unnecessary restarts by adjusting serialization order dynamically, and demonstrate that the new algorithm outperforms the previous ones over a wide range of system workload. It appears that this algorithm is a promising candidate for basic concurrency control mechanism for real-time database systems.

1. Introduction

Real-Time Database Systems (RTDBSs) differ from conventional database systems in a number of ways. In RTDBSs, transactions have timing constraints, the primary performance criterion is timeliness level and not average response time or throughput, and scheduling of transactions is driven by priority considerations rather than fairness considerations. Given these significant differences, considerable research has been recently devoted to designing concurrency control algorithms for RTDBSs and to evaluating their performance [1, 2, 7, 8, 9, 10, 11, 13, 14, 16]. Most of these algorithms are based on one of the two basic concurrency control mechanisms: *locking* [5] and *optimistic concurrency control* [12], and use priority information in the resolution of data conflicts, that is, resolve data conflicts in favor of the higher priority transaction.

The problem of scheduling transactions in an RTDBS with the objective of minimizing the percentage of transactions missing its deadline was first addressed in [1, 2]. Their work focused on evaluating the performance of various scheduling policies in RTDBSs through simulation experiments. A group of concurrency control algorithms for RTDBSs using two-phase locking as the underlying concurrency control mechanism was proposed and evaluated.

The study in [7, 8] focused primarily on the behavior of concurrency control protocols in a real-time database environment. The study showed that under the condition that tardy transactions are discarded from the system, optimistic concurrency control outperforms locking over a wide range of system loading and resource availability. The key reason for this result was described that the optimistic method, due to its validation stage conflict resolution, ensures that eventually discarded transactions do not restart other transactions unlike the locking approach in which soon-to-be-discarded transactions may restart other transactions. Such restarts are referred to as *wasted restarts* [7].

In [8], the problem of adding transaction timing information to optimistic concurrency control was addressed. They showed that the problem is nontrivial partly because giving preferential treatment to high priority transactions may result in an increase in the number of missed deadlines. In particular, the delayed conflict resolution policy of optimistic algorithms significantly reduces the possibility that a validating transaction sacrificed for an active transaction with a higher priority will meet its deadline. In addition, this situation may aggravate the real-time performance of the system in two more ways. One is that the validating transaction is restarted after spending most of the time and resources for its execution. The other is that there is no guarantee that the active transaction which caused the restart of the validating transaction will meet its deadline. If the active transaction does not meet its deadline for any reason, the sacrifice of the validating transaction is wasted. In [8], they studied several alternative schemes of incorporating transaction priority information into optimistic algorithms, including a scheme based on priority wait mechanism with wait control technique. However, this study and others [11, 14] showed that none of those schemes constantly outperforms the priority-incognizant algorithm.

The results of these studies suggest several implications. First, the choice of basic (priority-incognizant) concurrency control mechanism has significant impact on the performance of concurrency control for RTDBSs. The work in [7] showed that although the optimistic algorithm does not take transaction deadlines into account in making data conflict resolution decisions, it can still outperform a deadline-cognizant locking algorithm in a real-time database environment. Second, the number of restarts incurred by concurrency control is the major factor deciding the performance of concurrency control in real-time database systems, that is, having more restarts leads to poorer performance. Therefore restarts should be avoided if possible. In fact, the same result was derived in the studies of conventional database system performance [3]. Finally, more study should address the problem of designing deadline-sensitive optimistic

This work was supported in part by ONR, by DOE, and by IBM.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
1993		**00-00-1993 to 00-00-1993**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Virginia,Department of Computer Science,151 Engineer's Way,Charlottesville,VA,22904-4740**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**10**	

concurrency control algorithms. A practical real-time concurrency control algorithm should provide significant performance gains over the deadline-insensitive algorithms under a wide range of system workload and operating conditions.

In this paper, we address the problem of optimistic concurrency control algorithms that incur restarts unnecessary to ensure serializability among concurrent data access operations. We present a new optimistic concurrency control algorithm that can avoid such "unnecessary restarts," by dynamically adjusting serialization order of transactions through the use of timestamp intervals associated with running transactions. It appears that this protocol is a promising candidate for basic concurrency control mechanism for RTDBSs. The design of deadline-sensitive concurrency control schemes based on this algorithm for RTDBSs was discussed in a different paper [14].

The remainder of this paper is organized in the following fashion: Section 2 reviews the principle of optimistic concurrency control, and shows deficiencies of the optimistic methods used in previous studies. A new optimistic algorithm is presented in Section 3. Section 4 describes our real-time database environment for experiments. In Section 5, the results of the simulation experiments are highlighted. Finally, Section 6 summarizes the main conclusions of the study and outlines future study.

2. Optimistic Concurrency Control

In this section, we first discuss the principles underlying optimistic concurrency control, particularly regarding its validation. Then we show an example of unnecessary restarts incurred by the optimistic algorithms used in the previous studies.

2.1. Principles

In optimistic concurrency control, transactions are allowed to execute unhindered until they reach their commit point, at which time they are validated. Thus, the execution of a transaction consists of three phases: read, validation, and write [12]. The key component among these is the validation phase where a transaction's destiny is decided. Validation comes in several flavors, but every validation scheme is based on the following principle to ensure serializability.

If a transaction T_i is serialized before transaction T_j, the following two conditions must be satisfied:

Condition 1: *No overwriting*
The writes of T_i should not overwrite the writes of T_j.

Condition 2: *No read dependency*
The writes of T_i should not affect the read phase of T_j.

Generally, Condition 1 is automatically ensured in most optimistic algorithms because I/O operations in the write phase are required to be done sequentially in critical section. Thus most validation processes consider only Condition 2, and it can be carried out basically in either of the following two ways [6].

2.1.1. Backward Validation

In this scheme, the validation process is carried out against (recently) committed transactions. Data conflicts are detected by comparing the read set of the validating transaction and the write set of committed transactions, since it is obvious that committed transactions precede the validating transaction in serialization order. Such data conflicts should be resolved to ensure Condition 2. The only way to do this is to restart the validating transaction. The classical optimistic algorithm in [12] is based on this validation process.

Let T_v be the validating transaction and T_c ($c = 1, 2,..., n, c \neq v$) be the transactions recently committed with respect to T_v, i.e., those transactions that commit between the time when T_v starts executing and the time at which T_v enters the validation phase. Let $RS(T)$ and $WS(T)$ denote the read set and write set of transaction T, respectively. Then the backward validation operation can be described by the following procedure.

```
validate(T_v);
{
    valid := true;
    foreach T_c (c = 1, 2,..., n) {
        if WS(T_c) ∩ RS(T_v) ≠ { } then valid := false;
        if not valid then exit loop;
    }
    if valid then commit WS(T_v) to database
    else restart(T_v);
}
```

2.1.2. Forward Validation

In this scheme, validation of a transaction is done against currently running transactions. This process is based on the assumption that the validating transaction is ahead of every concurrently running transaction still in read phase in serialization order. Thus the detection of data conflicts is carried out by comparing the write set of the validating transaction and the read set of active transactions. That is, if an active transaction has read an object that has been concurrently written by the validating transaction, the values of the object used by the transactions are not consistent. Such data conflicts can be resolved by restarting either the validating transaction or the conflicting transactions in the read phase. Optimistic algorithms based on this validation process are studied in [6].

Let T_a ($a = 1, 2,..., n, a \neq v$) be the conflicting transactions in their read phase. Then the forward validation can be described by the following procedure.

```
validate(T_v);
{
    valid := true;
    foreach T_a (a = 1, 2, ... , n) {
        if RS(T_a) ∩ WS(T_v) ≠ { } then valid := false;
    }
    if valid then commit WS(T_v) to database
    else conflict resolution(T_v);
}
```

In real-time database systems, data conflicts should be resolved in favor of higher priority transactions. In backward validation, there is no way to take transaction priority into account

in the serialization process, since it is carried out against already committed transactions. Thus backward validation is not amenable to real-time database systems. Forward validation provides flexibility for conflict resolution that either the validating transaction or the conflicting active transactions may be chosen to restart, so it is preferable for real-time database systems. In addition to this flexibility, forward validation has the advantage of early detection and resolution of data conflicts.

All the optimistic algorithms used in the previous studies of real-time concurrency control in [7, 8, 9, 11, 13, 14] are based on the forward validation. The broadcast mechanism in the algorithm, OPT-BC used in [7, 8, 9], is an implementation variant of the forward validation. From now on, we refer to this algorithm as OCC-FV.

2.2. Unnecessary Restarts

As we mentioned above, forward validation is based on the assumption that the serialization order among transactions is determined by the arriving order of transactions at validation phase. Thus the validating transaction, if not restarted, always precedes concurrently running active transactions in serialization order. We claim that this assumption is not only unnecessary, but also the validation process based on this assumption can incur restarts not necessary to ensure data consistency. These restarts should be avoided. To ensure this claim, let us consider the following example.

Example 1: Let $r_i[x]$ and $w_i[x]$ denote a read and write operation, respectively, on the data object x by transaction i, and let v_i and c_i denote the validation and commit of transaction i, respectively. Consider three transactions T_1, T_2, and T_3:

T_1: $r_1[x]$ $w_1[x]$ $r_1[y]$ $w_1[y]$ v_1
T_2: $r_2[x]$ $w_2[x]$ ... v_2
T_3: $r_3[y]$ ... v_3

and suppose they execute as follows:

$H_1 = r_1[x]\ w_1[x]\ r_2[x]\ r_3[y]\ w_2[x]\ r_1[y]\ w_1[y]\ v_1\ c_1$.

If we use the forward validation process described above for the validation of T_1, both the active transactions, T_2 and T_3 are conflicting with T_1 on data items x and y, respectively, and should restart. It is fair for T_2 to restart since it has both write-write and write-read conflicts with T_1. However, T_3, we observe, does not have to restart, if there is no more conflict with T_1 than the write-read conflict on data item y. In fact no serialization order between T_1 and T_3 has been built except for the read-write conflict on y. If we set the serialization order between T_1 and T_3 as $T_3 \rightarrow T_1$ during the validation of T_1, we can ensure data consistency without restarting T_3.■

We refer to such a restart of T_3 in the forward validation as an *unnecessary restart*. Also, we refer to transactions having both write-write and write-read conflicts with the validating transaction like T_2 as *irreconcilably conflicting*, while transactions having only write-read conflicts like T_3 as *reconcilably conflicting*.

The design of the new optimistic algorithm presented in the next section is based on this categorization of active transactions. As we will explain, the categorization is automatically done by adjusting and recording the current serialization order dynamically using timestamp intervals associated with each active transaction. The performance gain by the new algorithm can be significant especially when reconcilable conflicts dominate, that is, the probability that a data object read is updated is low, which is true for most actual database systems. Generally, under a wide range of system workload, the algorithm provides a performance advantage by reducing the number of restarts at the expense of maintaining serialization order dynamically.

3. A New Optimistic Algorithm

In this section, we present the proposed optimistic algorithm in detail. We first explain the mechanism to guarantee serializability used in the algorithm, and then prove its correctness. At the end, we discuss the advantages and disadvantages of this protocol. We hereafter refer to this algorithm as OCC-TI.

3.1. Validation Phase

In this protocol, every transaction in the read phase is assigned an timestamp interval, which is used to record temporary serialization order induced during the execution of the transaction. At the start of execution, the timestamp interval of a transaction is initialized as $[0,\infty)$, i.e., the entire range of timestamp space. Whenever serialization order of a transaction is induced by its data operation or the validation of other transactions, its timestamp interval is adjusted to represent the dependencies. In addition to the timestamp interval, a final timestamp is assigned to each transaction which has successfully passed its validation test and guaranteed to commit.

In this algorithm, a transaction that finishes read phase and reaches validation is always guaranteed to commit as in OCC-FV. However, unlike OCC-FV in which a transaction is validated by comparing its write set and the read sets of transactions, the validation of a transaction in OCC-TI consists of adjusting the timestamp intervals of concurrent transactions.

Let $TI(T)$ and $TS(T)$ denote the timestamp interval and final timestamp of transaction T, respectively. Then the validation process can be briefly described by the following procedure.

```
validate(T_v);
{
    select TS(T_v) from TI(T_v);
    foreach T_a (a = 1, 2, ... , l) {
        adjust(T_a);
    }
    foreach D_i (i = 1, 2, ... , m) in RS(T_v) {
        update RTS(D_i);
    }
    foreach D_j (j = 1, 2, ... , n) in WS(T_v) {
        update WTS(D_j);
    }
    commit WS(T_v) to database;
}
```

First, the final timestamp, $TS(T_v)$, is determined from the timestamp interval, $TI(T_v)$. In fact, any timestamp in $TI(T_v)$ can be chosen to be $TS(T_v)$, because any value in $TI(T_v)$ preserves the serialization order induced by T_v. In this algorithm, we always select the minimum value of $TI(T_v)$ for $TS(T_v)$ for a practical reason, which will be made clear later. $TS(T_v)$ is used in the next steps of the validation process. Second, the timestamp intervals of all the concurrently running transactions that have accessed common objects with T_v are adjusted to reflect the serialization order induced between T_v and those transactions. Any active transaction whose timestamp interval shuts out by the adjustment operation should restart, because it has introduced nonserializable execution with respect to T_v. The details of the adjustment procedure will be described below. Finally, if necessary, the final timestamp of the committing transaction is recorded for every data object it has accessed. *RTS(D)* and *WTS(D)* denote the largest timestamp of committed transactions that have read and written, respectively, data object *D*. The need of this operation will be explained in the next section.

The salient point of OCC-TI is that unlike other optimistic algorithms, it does not depend on the assumption of the serialization order of transactions being the same as the validation phase arriving order, but it records serialization order induced precisely and uses restarts only when necessary. Let us examine how serialization order is adjusted between the validating transaction and a concurrently active transaction for the three possible types of conflict:

Read-write conflict ($RS(T_v) \cap WS(T_a) \neq \{\}$)

This type of conflicts leads the serialization order between T_v and T_a to $T_v \rightarrow T_a$. That is, the timestamp interval of T_a is adjusted to follow that of T_v. We refer to this ordering as *forward ordering*. The implication of this ordering is that the read phase of T_v is not affected by the writes of T_a.

Write-read conflict ($WS(T_v) \cap RS(T_a) \neq \{\}$)

In this case, the serialization order is recorded as $T_a \rightarrow T_v$. That is, the timestamp interval of T_a is adjusted to precede that of T_v. This ordering is referred to as *backward ordering*. It implies that the writes of T_v have not affected the read phase of T_a.

Write-write conflict ($WS(T_v) \cap WS(T_a) \neq \{\}$)

A write-write conflict results in forward ordering, i.e., $T_v \rightarrow T_a$. Thus the order implies that T_v's writes do not overwrite T_a's writes.

Non-serializable execution is detected when the timestamp interval of an active transaction shuts out. The non-serializable execution is deleted from execution history by restarting the transaction. Obviously, the timestamp interval of an active transaction that requires both backward and forward ordering to record the execution of its operations will shut out. Such transaction are irreconcilably conflicting with the validating transaction.

The adjustment of timestamp intervals of active transactions is the process of recording serialization order according to the conflict types and their corresponding ordering. It can be described by the following procedure. We assume that timestamp intervals contain only integers.

```
adjust(T_a);
{
    foreach D_i (i = 1, 2, ... , m) in RS(T_v) {
        if D_i in WS(T_a)
        then TI(T_a) := TI(T_a) ∩ [TS(T_v), ∞);
        if TI(T_a) = [ ] then restart(T_a);
    }

    foreach D_j (j = 1, 2, ..., n) in WS(T_v) {
        if D_j in RS(T_a)
        then TI(T_a) := TI(T_a) ∩ [0, TS(T_v)-1];
        if D_j in WS(T_a)
        then TI(T_a) := TI(T_a) ∩ [TS(T_v), ∞);
        if TI(T_a) = [ ] then restart(T_a);
    }
}
```

3.2. Read Phase

The adjustment of active transactions' timestamp intervals at the validation of a transaction is the process of recording the serialization order between the committing transaction and the data operations performed by the concurrently running transactions until the moment. Because the active transactions continue to execute remaining data operations, the execution order induced by the remaining operations should be checked to determine if they induce any non-serializable execution. If so, the active transaction should restart. The following example demonstrates such late restart.

Example 2: Consider two transactions T_1 and T_2:

T_1: $r_1[y]$ $w_1[y]$ v_1
T_2: $r_2[y]$ $w_2[y]$ ... v_2

and suppose they execute as follows:

$H_2 = r_1[y]\ r_2[y]\ w_1[y]\ v_1\ c_1\ w_2[y]$ ...

At the validation of T_1, T_2 has only a write-read conflict with T_1. With the backward ordering, the serialization order between T_1 and T_2 is set as $T_2 \rightarrow T_1$, and T_2 is not restarted. However, later the write operation of T_2, $w_2[y]$, induces a serialization order between T_1 and T_2 in opposite direction. Thus T_2 has to restart.■

The detection of non-serializable execution by remaining operations of active transactions can also be done using the timestamp intervals. Because, in this case, the serialization order of active transactions is checked against committed transactions, we need to use the timestamps of data objects, i.e., *RTS(D)* and *WTS(D)* of data object *D*. In the read phase, whenever a transaction performs a data operation, its timestamp interval is adjusted to reflect the serialization induced between the transaction and committed transactions. If the timestamp interval shuts out, a non-serializable execution performed by the transaction is detected, and the transaction restarts. The process can be described by the following procedure.

```
read_phase(T_a);
{
    foreach D_i (i = 1, 2, ..., m) in RS(T_a) {
        read(D_i);
        TI(T_a) := TI(T_a) ∩ [WTS(D_i), ∞);
        if TI(T_a) = [ ] then restart(T_a);
    }

    foreach D_j (j = 1, 2, ..., n) in WS(T_a) {
    pre-write(D_j);
    TI(T_a) := TI(T_a) ∩ [WTS(D_j), ∞) ∩ [RTS(D_j), ∞);
    if TI(T_a) = [ ] then restart(T_a);
    }
}
```

Example 3: To understand how this procedure works, let us consider the previous example again. The execution history is given as follows:

$$H_2 = r_1[y]\ r_2[y]\ w_1[y]\ v_1\ c_1\ w_2[y].$$

At its validation, T_1 is first assigned a final timestamp $TS(T_1)$, say 74. Then with backward ordering, the timestamp interval of T_2 is adjusted to be [0, 73]. In addition, the timestamps of data object, $RTS(y)$ and $WTS(y)$, accessed by T_1 are updated to be 74. After the validation process, when T_2 performs $w_2[y]$, its timestamp is adjusted by the following operation:

$$TI(T_2) := [0, 73] \cap [74, \infty) \cap [74, \infty).$$

Because this operation leaves $TI(T_2)$ shut out, non-serializable execution is detected and T_2 restarts.■

Note that in OCC-FV, transactions in read or validation phase do not need to check for conflicts with already committed transactions. In this algorithm, transactions conflicting with a committed transaction would have been restarted earlier by the committed transaction [7].

3.3. Write Phase

Once a transaction is in the write phase, it is considered to be committed. All committed transactions can be serialized by the final timestamp order. In the write phase, the only work of a transaction is making all its updates permanent in the database. Data objects are copied from the local workspace into the database. Since a transaction applies the results of its write operations only after it commits, the *strictness* [4] of the histories produced by OCC-TI is guaranteed. This property makes the transaction recovery procedure simpler than non-strict concurrency control protocols.

3.4. Correctness

In this section, we give an argument on the correctness of the algorithm. First, we give simple definitions of history and serialization graph (*SG*). The formal definitions for these concepts can be found in [4]. A history is a partial order of operations that represents the execution of a set of transactions. Any two conflicting operations must be comparable. Let H denote a history. The serialization graph for H, denoted by $SG(H)$, is a directed graph whose nodes are committed transactions in H and whose edges are all $T_i \rightarrow T_j$ $(i \neq j)$ such that one of T_i's operations precedes and conflicts with one of T_j's operations in H. To prove a history H serializable, we only have to prove that $SG(H)$ is acyclic [4].

Lemma 1: Let T_1 and T_2 be two committed transactions in a history H produced by the proposed algorithm. If there is an edge $T_1 \rightarrow T_2$ in $SG(H)$, then $TS(T_1) < TS(T_2)$.

Proof: Since there is an edge, $T_1 \rightarrow T_2$ in $SG(H)$, the two must have one or more conflicting operations whose type is one of the following three:

Case 1: $r_1[x] \rightarrow w_2[x]$

This case implies that T_1 commits before T_2 reaches its validation phase since $r_1[x]$ is not affected by $w_2[x]$. For $w_2[x]$, OCC-TI adjusts $TI(T_2)$ to follow $RTS(x)$ that is equal to or is greater than $TS(T_1)$. That is, $TS(T_1) \leq RTS(x) < TS(T_2)$. Therefore, $TS(T_1) < TS(T_2)$.

Case 2: $w_1[x] \rightarrow r_2[x]$

This case is possible only when the write phase of T_1 finishes before $r_2[x]$ executes in T_2's read phase. For $r_2[x]$, OCC-TI adjusts $TI(T_2)$ to follow $WTS(x)$, which is equal to or greater than $TS(T_1)$. That is, $TS(T_1) \leq WTS(x) < TS(T_2)$. Therefore, $TS(T_1) < TS(T_2)$.

Case 3: $w_1[x] \rightarrow w_2[x]$

This case can be similarly proved to lead to $TS(T_1) < TS(T_2)$.■

Theorem: Every history generated by OCC-TI algorithm is serializable.

Proof: Let H denote any history generated by the algorithm. Suppose, by way of contradiction, that $SG(H)$ contains a cycle $T_1 \rightarrow T_2 \rightarrow \ ... \rightarrow T_n \rightarrow T_1$, where $n > 1$. By Lemma 1, we have $TS(T_1) < TS(T_2) < ... < TS(T_n) < TS(T_1)$. This is a contradiction. Therefore no cycle can exist in $SG(H)$ and thus the algorithm produces only serializable histories.■

3.5. Discussion

In this section, we discuss the advantages and disadvantages of OCC-TI, and consider ways to include transaction deadline information in making conflict resolution decisions.

The algorithm keeps all the advantages of OCC-FV. They include high degree of concurrency, freedom from deadlock, early detection and resolution of conflicts (compared to backward validation-based optimistic algorithm) resulting in both less wasted resources and earlier restarts. All of these contribute to increasing the chances of meeting transaction deadlines. Also, like the OCC-FV algorithm, OCC-TI can avoid the "wasted restart" phenomenon of locking-based algorithms because it allows only committed transactions to restart others. Furthermore, the ability of OCC-TI to avoid unnecessary restarts is expected to provide performance gains over the OCC-FV algorithm.

However, this expected performance gain does not come for free. The main cost for this benefit is the management of

timestamp intervals for active transactions and timestamps for data objects. This management can be done efficiently by using a transaction table and a data object table. The transaction table contains information of active transactions, including the read set and write set, and the timestamp interval of every active transaction. The information that is recorded in the data object table includes for each data object, *D*, *WTS(D)*, *RTS(D)*, the list of transactions holding locks on *D*, and the waiting list of incompatible lock requests on *D*. The data object table is also called a lock table (see Section 4.5.).

Another important point to note here is that the degree of performance gain due to avoiding unnecessary restarts is dependent on the probability that a data object read is updated. When this *write probability* is low, that is, a write-read conflict rarely leads to a write-write conflict, the performance advantage can be significant. However, if the write probability is high, that is, a backward ordering for write-read conflict is almost always followed by a forward ordering for write-write conflict, the cost for timestamp interval management can overwhelm the benefit of reduced number of restarts.

Finally, it should be noted that OCC-TI presented in this paper does not use any transaction deadline information to make decisions for conflict resolution. The incorporation of timing information into the algorithm to improve timeliness level is a problem to be addressed. One method to do this was studied in [13, 14]. In this method, at the validation of a transaction, the set of active transactions is divided into two groups: reconcilably conflicting set and irreconcilably conflicting set. Conflict resolution between the validating transaction and the active transactions in the irreconcilable set is done by a deadline-sensitive scheme called *Feasible Sacrifice* [14]. Then the timestamp interval adjustment process for reconcilable set follows only when the validating transaction is decided to commit by the priority-based conflict resolution.

The Feasible Sacrifice scheme gives precedence to urgent transactions, while reducing the number of missed deadlines due to wasted sacrifices through the use of a feasibility test of every validating transaction. For the feasibility test, we proposed an approach to estimating the execution time of restarted transactions in optimistic protocols. In [14], the Feasible Sacrifice scheme is shown to provide significant gains over deadline-insensitive optimistic algorithms, and to outperform constantly the conflict resolution scheme based on a priority wait mechanism with a wait control technique [8].

4. Experiment Environment

This section outlines the structure and details of our simulation model and experimental environment which were used to evaluate the performance of concurrency control algorithms for RTDBSs. The issues on the implementation of optimistic schemes are also discussed.

4.1. Tardy Transaction Policy

It has been shown that the policy dealing with tardy transactions has a significant impact on the relative performance of the concurrency control algorithms in real-time database systems [7]. Different policies for tardy transactions are needed for real-time applications with *soft deadlines* and *firm deadlines*. In the former case, tardy transactions may have to run to completion (maybe with promoted priority), and in the latter case, they are considered having lost all value and hence be discarded from the system. Examples of applications having these types of deadline are given in [1]. The differences of the performance behavior of these two policies have been examined using simple queueing systems in [7]. In the experiments in this paper, we assume that transactions arriving in the system have firm deadlines. Therefore, transactions are discarded immediately after they miss deadline.

4.2. Simulation Model

Central to our simulation model for RTDBS is a single-site disk resident database system operating on shared-memory multiprocessors. The physical queueing model is depicted in Figure 1, and the associated model parameters are described in the next section. The physical queueing model is similar to the one used in [3]. CPUs share a single queue and the service discipline used for the queue is priority scheduling without preemption. Each disk has its own queue and is also scheduled with priority scheduling.

In this model, the execution of a transaction consists of multiple instances of alternating data access request and data operation steps until all of the data operations in it complete or it is aborted for some reason. When a transaction makes a data access request, the request must go through concurrency control to obtain a permission to access the data object. If the request is granted, the transaction proceeds to perform the data operation which consists of a disk access and CPU computation. The transaction passes through disk queue and CPU queue. If the data access request is denied, the transaction will be placed into a data queue. The waiting transaction will be awakened when the requested data object becomes available.

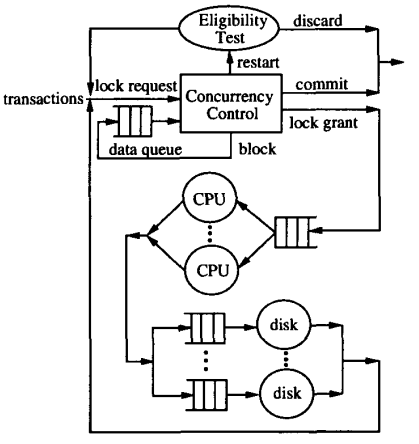

Figure 1 Simulation Model

If a data access request leads to a decision to abort the transaction, it has to restart. The system checks the eligibility of a transaction whenever it restarts, and whenever it is put into and comes out of a queue, to see if it has already missed its deadline. With the firm deadline assumption, transactions that has missed deadline are aborted and permanently discarded from the system.

4.3. Parameter Setting

The database itself is modeled as a collection of data pages in disks, and the data pages are modeled as being uniformly distributed across all the disks. A transaction consists of a mixed sequence of read and write operations. We assume that a write operation is always preceded by a read for the same page, that is, the write set of a transaction is always a subset of its read set [4].

Table 1 gives the names and meanings of the parameters that control system resources. The parameters, *CPUTime* and *DiskTime* capture the CPU and disk processing times per data page. Our simulation system does not account for the time needed for lock management and context switching. We assume that these costs are included in *CPUTime* on a per data object basis.

The use of database buffer pool is simulated using probability, rather than each buffer page being traced individually. When a transaction attempts to read a data page, the system determines whether the page is in memory or disk using the probability, *BufProb*. If the page is determined to be in memory, the transaction can continue processing without disk access. Otherwise, an IO service request is created.

Table 2 summarizes the key parameters that characterize system workload and transactions. Transactions arrive in a Poisson stream, i.e., their inter-arrival times are exponentially distributed. The *ArriRate* parameter specifies the mean rate of transaction arrivals. The number of data objects accessed by a transaction is determined by a normal distribution with mean *TranSize*, and the actual data objects are determined uniformly from the database. A page that is read is updated with the probability, *WriteProb*.

Table 1: Workload Parameters

Parameter	Meaning	Base Value
ArriRate	Mean transaction arrival rate	-
TranSize	Average transaction size (in pages)	10
WriteProb	Write probability per accessed page	0.25
MinSlack	Minimum slack factor	2
MaxSlack	Maximum slack factor	8

Table 2: System Resource Parameters

Parameter	Meaning	Base Value
DBSize	Number of data pages in database	400
NumCPUs	Number of processors	2
NumDisks	Number of disks	4
CPUTime	CPU time for processing a page	15 msec
DiskTime	Disk service time for a page	25 msec
BufProb	Prob. of a page in memory buffer	0.5

The assignment of deadlines to transactions is controlled by the parameters, *MinSlack* and *MaxSlack*, which set a lower and upper bound, respectively, on a transaction's slack time. We use the following formula for deadline-assignment to a transaction:

$$Deadline = AT + \text{uniform}(MinSlack, MaxSlack) * ET.$$

AT and *ET* denote the arrival time and execution time, respectively. The execution time of a transaction used in this formula is not an actual execution time, but a time estimated using the values of parameters, *TranSize*, *CPUTime* and *DiskTime*. This value is used only for the above deadline-assignment formula, but not used for any other purpose including conflict resolution decisions in concurrency control. In this system, the priorities of transactions are assigned by the *Earliest Deadline First* policy [15], which uses only deadline information to decide transaction priority, but not any other information about transaction execution time.

Finally, the base values for parameters shown in Tables 1 and 2 are not meant to model a specific real-time application. They were chosen to be reasonable for a wide range of actual database systems.

4.4. Performance Metrics

The primary performance metric used is the percentage of transactions which miss their deadlines, referred to as *Miss Percentage*. *Miss Percentage* is calculated with the following equation:

$$Miss\ Percentage = 100 * (\#\ \text{of tardy jobs} / \#\ \text{of jobs arrived}).$$

In addition to *Miss Percentage*, we measure other statistical information, including system throughput, average number of transaction restarts, average data blocking time, and average resource queueing time. These secondary measures help explain the behavior of the concurrency control algorithms under various operating conditions. The average number of transaction restarts, which we refer to as *Restart Count*, is normalized on a per-transaction basis, so that its value represents the average number of restarts experienced by a transaction until it completes, or misses its deadline and is discarded.

4.5. Implementation Issues

In this section, we describe a physical implementation method for optimistic algorithms with forward validation described at the logical level earlier in this paper. The implementation is one of the major conditions for a correct performance comparison of concurrency control algorithms.

For the implementation, we have two major concerns. One is the *efficiency* of validation. At the logical level, data conflicts are detected comparing the read set and write set of transactions. This method is impractical for actual database systems, since the complexity of the validation test is dependent on the number of active transactions. The other concern is the *comparability* of locking algorithms and optimistic schemes. In previous studies of concurrency control for RTDBSs in [7, 8], the validation test of optimistic algorithms were implemented using broadcast mechanism by which the validating transaction notifies

other currently running transactions with data conflicts. The concept is straightforward, but it is difficult to compare the performance of locking protocol with optimistic algorithm implemented using broadcast mechanism due to the significant difference in their implementation. It is difficult to determine the fair cost for each implementation mechanism. This is especially true for simulation study, and also applies to performance study using actual systems such as testbed, because the implementation costs often vary from one system to another.

Based on these reasons, we decided to use a locking mechanism for the implementation of optimistic protocols. The mechanism is based on the one proposed in [11]. In this mechanism, the system maintains a system-wide lock table, LT, for recording data accesses by all concurrently executing transactions. There are two lock modes - read-phase lock (*R-lock*) and validation-phase lock (*V-lock*). An R-lock is set in LT whenever a transaction accesses a data object in its read phase, and an R-lock for write operation is upgraded to a V-lock when the transaction enters its validation phase. The two lock modes are not compatible. Generally, the validation process is carried out by checking the lock compatibility with the lock table. This locking-based implementation of validation test provides both efficiency and implementation comparability due to its complexity independent of the number of active transactions, and its use of locking, respectively.

It is shown in [11] that the physical operations on the lock table of this implementation of the optimistic protocol are almost the same as those of locking protocols. Despite the similarity, there are some differences that may affect the relative performance of the locking and optimistic protocols. First, the locking duration of optimistic algorithm is shorter than that of locking protocols, since it is only during the validation and write phase of a transaction. Second, the R-locks of optimistic protocol will not block any concurrent transactions in the read phase. Finally, the optimistic protocol maintains the property of deadlock-freedom, even though R-locks and V-locks are used.

5. Experiments and Results

In this section, we present performance results from our experiments comparing concurrency control algorithms in a real-time database system. We compare three different concurrency control protocols: 2PL-HP which is basically a locking scheme, but resolves a data conflict between a lower priority lock holder and a higher priority lock requester by restarting the lower priority transaction [1], OCC-FV [7], and OCC-TI. Note again that OCC-FV and OCC-TI do not use transaction deadline information for data conflict resolution, while 2PL-HP does.

The simulation program was written in SIMAN, a discrete-event simulation language [17]. The data collection in the experiments is based on the method of replication. For each experiment, we ran the simulation with the same parameter values for at least 10 different random number seeds. Each run continued until 1,000 transactions were executed. For each run, the statistics gathered during the first few seconds were discarded in order to let the system stabilize after initial transient condition. The statistical data reported in this paper has 90% confidence intervals whose end points are within 10% of the point estimate. In the following graphs, we only plot the mean values of the performance metrics.

First, we examine the performance of concurrency control algorithms under the condition of limited resources. The values of parameters, *NumCPUs* and *NumDisks* are fixed two and four, respectively. Figure 2 shows *Miss Percentage* behavior of algorithms under different levels of system workload. System workload is controlled by the arrival rate of transactions in the system. In this experiment, the value of *WriteProb* is fixed at 0.25. From the graph, it is clear that for very low arrival rates under 10 transactions/second, there is not much difference for the three protocols. However, as the arrival rate increases, OCC-FV does progressively better than 2PL-HP, and OCC-TI does even better than OCC-FV.

One of the reasons for this performance difference is the difference in the number of restarts, *Restart Count*, incurred by each of the protocols, shown in Figure 3. As mentioned earlier, 2PL-HP suffers performance degradation caused by wasted restarts. That is, the immediate conflict resolution policy of 2PL-HP allows a transaction that will eventually miss its deadline and be discarded to restart (or block) other transactions. This performance degradation increases as the workload level increases, since the number of transactions that miss their deadlines and have to be discarded increases.

The delayed conflict resolution policy of optimistic algorithms helps them to avoid such wasted restarts. However, OCC-FV suffers performance degradation caused by unnecessary restarts described in Section 3. At the relatively low write probability of 0.25, the possibility of a backward ordering followed by a forward ordering is low, and many unnecessary restarts can be saved by OCC-TI protocol. This is shown clearly in Figure 3, where we observe a significant difference between the restart curves of OCC-FV and OCC-TI.

Figures 4 and 5 show similar graphs as Figures 2 and 3, i.e., *Miss Percentage* and *Restart Count* of the these protocols under different levels of system workload. In this case, however, the system operates at a higher level of data contention with the value of *WriteProb* fixed at 0.75. The performance difference between 2PL-HP and OCC-FV becomes even bigger, since the number of wasted restarts in 2PL-HP tends to increase as data contention increases. However, OCC-TI does not show significant performance gains over OCC-FV. This is due to the fact that with the relatively high write probability of 0.75, not many restarts are made unnecessarily by OCC-FV, since most backward ordering is followed by forward ordering.

One point to note here is that the restart count of all the three protocols decreases after a certain workload. The reason for this decrease is that after that workload point, resource contention dominates data contention in discarding deadline-missing transactions.

Until now, the performance of the protocols was shown under a limited resource situation. In such situations, since resource contention dominates data contention quickly as system workload increases, the performance of the system is primarily determined by resource scheduling algorithms rather than

concurrency control algorithms. In fact, the performance difference shown in Figures 2 and 4 may not be very striking.

To capture the performance difference of concurrency control algorithms without the effect of resource contention, we simulated an infinite resource situation, where there is no queueing for resources (CPUs and disks). The data contention is maintained relatively high with the value of *WriteProb* fixed at 0.5. Figures 6 and 7 show *Miss Percentage* and *Restart Count* of the three protocols. As we expected, the performance difference of the three becomes clearer. Note that since in this infinite resource situation, there is no resource contention, the restart counts of the three protocols ever increase as the system workload increases.

6. Conclusions

In this paper, we have presented a new optimistic concurrency control algorithm. The design of the algorithm was motivated by the recent study results in [7, 8] concluding that optimistic approach outperforms locking protocols in real-time database systems with the objective of minimizing the percentage of transactions missing deadline. We observed that the optimistic algorithms used in those studies could incur restarts unnecessary to ensure data consistency. The new optimistic algorithm was designed to precisely adjust and record temporary serialization order among concurrently running transactions, and thereby to avoid unnecessary restarts. To evaluate the effect of the unnecessary restarts, a quantitative study was carried out using a simulation system of RTDBS with three concurrency control algorithms: two-phase locking with high priority conflict resolution policy (2PL-HP), optimistic protocol with forward validation (OCC-FV) and the proposed optimistic algorithm (OCC-TI).

We showed that under the policy that discards tardy transactions from the system, the optimistic algorithms outperform 2PL-HP, and OCC-TI does better than OCC-FV among the optimistic algorithms. The performance difference between OCC-FV and OCC-TI becomes large especially when the probability of a data object read being updated is low, which is true in most actual database systems. In conclusion, the factor of unnecessary restarts is not negligible in performance of optimistic concurrency control under both finite and infinite resource conditions, and the proposed optimistic algorithm is a promising candidate for basic concurrency control mechanisms for real-time database systems.

The optimistic algorithm proposed in this paper does not use transaction deadline information in making decisions for data conflict resolution. We expect a better concurrency control algorithm by using an intelligent way of incorporating transaction deadline information into the basic mechanism. In a different paper [14], we studied deadline-sensitive concurrency control mechanisms based on OCC-TI for RTDBSs, and proposed a protocol that outperforms other real-time concurrency control algorithms currently known under a wide range of operating conditions.

REFERENCE

[1] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proceedings of the 14th VLDB Conference*, Aug. 1988.

[2] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data," *Proceedings of the 15th VLDB Conference*, Aug. 1989.

[3] Agrawal, R., M. J. Carey, and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Trans. on Database Systems*, Dec. 1987.

[4] Bernstein, P. A., V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, MA 1987.

[5] Eswaran, K., J. Gray, R. Lorie, and I. traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Comm. of ACM*, Nov. 1976.

[6] Haerder, T., "Observations on Optimistic Concurrency Control Schemes," *Information Systems*, 9(2), 1984.

[7] Haritsa, J. R., M. J. Carey, and M. Livny, "On Being Optimistic about Real-Time Constraints," *Proceedings of the 1990 ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems (PODS)*, 1990.

[8] Haritsa, J. R., M. J. Carey, and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," *Proceedings of the 11th IEEE Real-Time Systems Symposium*, Orlando, Florida, Dec. 1990.

[9] Haritsa, J. R., M. J. Carey, and M. Livny, "Data Access Scheduling in Firm Real-Time Database Systems," *The Journal of Real-Time Systems*, Kluwer Academic Publishers, 4, 1992.

[10] Huang, J., J. A. Stankovic, D. Towsley and K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing," *Proceedings of the 10th IEEE Real-Time Systems Symposium*, Dec. 1989.

[11] Huang, J., J. A. Stankovic, K. Ramamritham, and D. Towsley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes," *Proceedings of the 17th VLDB Conference*, Sep. 1991.

[12] Kung H. T., and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, June 1981.

[13] Lee, J., and S. H. Son, "An Optimistic Concurrency Control Protocol for Real-Time Database Systems," *3rd International Symposium on Database Systems for Advanced Applications*, Daejon, Korea, April 1993.

[14] Lee, J. and S. H. Son, "Design of Optimistic Concurrency Control for Real-Time Database Systems," submitted for publication.

[15] Liu, C. L., and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J.ACM* 10(1), 1973.

[16] Lin Y., and S. H. Son, "Concurrency Control in Real-Time Database Systems by Dynamic Adjustment of Serialization Order," *Proceedings of the 11th IEEE Real-Time Systems Symposium*, Orlando, Florida, Dec. 1990.

[17] Pegden, C. D., R. Shannon, and R. Sadowski, *Introduction to Simulation Using SIMAN*, McGraw-Hill, Inc., NJ, 1990.

Figure 2 Miss Percentage, Write Probability = 0.25
Finite Resource, Tardy Transactions Discarded

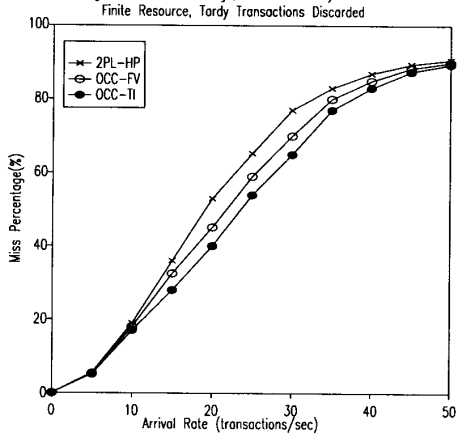

Figure 3 Restart Count, Write Probability = 0.25
Finite Resource, Tardy Transactions Discarded

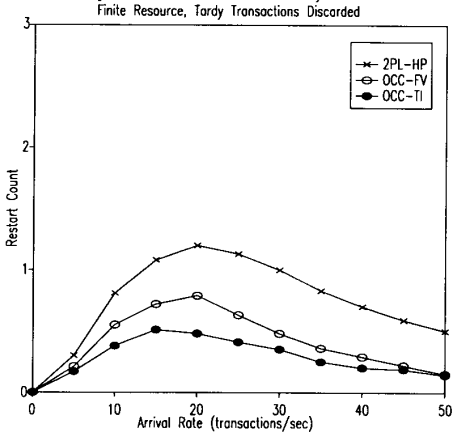

Figure 4 Miss Percentage, Write Probability = 0.75
Finite Resource, Tardy Transactions Discarded

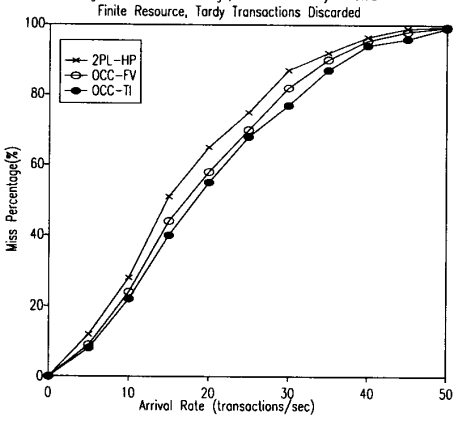

Figure 5 Restart Count, Write Probability = 0.75
Finite Resource, Tardy Transactions Discarded

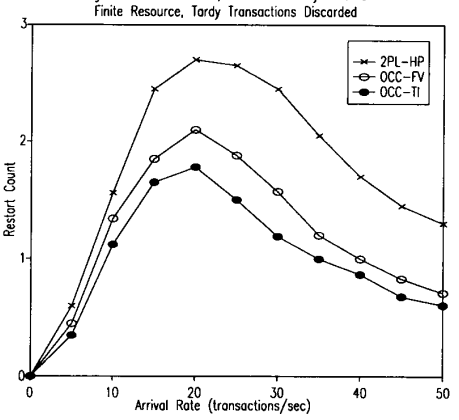

Figure 6 Miss Percentage, Write Probability = 0.5
Infinite Resource, Tardy Transactions Discarded

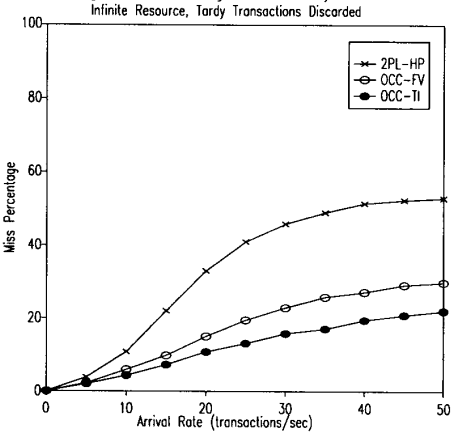

Figure 7 Restart Count, Write Probability = 0.5
Infinite Resource, Tardy Transactions Discarded